

Algorithms in Systems Engineering IE170

Final Review

Dr. Ted Ralphs

Textbook Sections Covered in Course

Introduction to Part I

Chapter 1, all sections

Chapter 2, all sections

Chapter 3, all sections

Chapter 4, intro and sections 1-3

Introduction to Part II

Chapter 6, all sections

Chapter 7, all sections

Introduction to Part III

Chapter 10, all sections

Chapter 11, intro and sections 1-4

Chapter 12, all sections

Chapter 21, intro and sections 1-3

Introduction to Part VI

Chapter 22, intro and sections 1-4.

Chapter 23, intro and section 3.

Chapter 28, intro and sections 1 and 3.

Chapter 31, intro and sections 2, 4, 6, 7, 8.

Chapter 32, intro and sections 1, 2, and 4.

Algorithms

Algorithms

- The main theme of the course has been the **design, implementation, and analysis of algorithms and data structures** for **solving problems**.
- A *problem* specifies the form of the output desired for a given set of inputs.
- An algorithm is a specific procedure for converting input to output.
- An algorithm is said to be *correct* for a given problem if it successfully converts each possible input into an output of the desired form, called a *solution*.
- Ultimately, we are interested in algorithms that are **fast**.
- We will judge the speed of an algorithm by the number of **fundamental operations** required to execute it, generally called the *running time*.
- This provides a measure that is independent of **hardware**.

Analyzing Algorithms

- The goal of analyzing an algorithm is to determine its running time.
- A *problem instance* is just a specific set of inputs.
- The running time of an algorithm is different for different instances.
- This creates difficulties for analysis and leads to two different summary measures of running time.
 - Worst-case
 - Average-case
- Worst-case is almost always easier to compute, but average-case is often a more useful measure.
- We can use either theoretical or empirical analysis, or a combination, to determine running time.

Models of Computation

- In order to analyze the number of steps necessary to execute an algorithm, we have to say what we mean by a “step.”
- To define this precisely is tedious and beyond the scope of this course.
- A precise definition depends on the exact hardware being used.
- Our analysis will assume a very simple model of a computer called a *random access machine* (RAM).
- In a RAM, the following operations take one step.
 - **arithmetic** (addition, subtraction, multiplication, division)
 - **data movement** (read from memory, store in memory, copy)
 - **comparison**
 - **control** (function calls, goto commands)
- This is a very idealized model, but it works in practice.
- We will sometimes need to simplify the model even further.

The Input Size

- The running time of an algorithm generally depends primarily on the number of input values, or the *size of the input*.
- We are interested in how the running time grows generally as the input size grows.
 - Because we are mainly interested in how the running time grows as the instances become larger, we won't need “exact” running times.
 - We will allow some “sloppiness” and ignore constants and low order terms.
 - Because of our many simplifying assumptions, the low order terms may not be accurate anyway.
- The growth rate of algorithms gives us a basis for comparison.
 - Any algorithm can be used to solve a small problem.
 - It is the really large problems that require efficient algorithms.

Growth of Functions

- Consider algorithm A with running time given by f and algorithm B with running time given by g .
- We are interested in knowing

$$L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

- There are four possibilities.
 - $L = 0$: g grows faster than f .
 - $L = \infty$: f grows faster than g .
 - $L = c$: f and g grow at the same rate.
 - The limit doesn't exist.

Θ Notation

- We now define the set

$$\Theta(g) = \{f : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n) \forall n \geq n_0\}$$

- If $f \in \Theta(g)$, then we say that f and g *grow at the same rate* or that they are *of the same order*.
- Note that

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

- We also know that if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ for some constant c , then $f \in \Theta(g)$.
- If the limit doesn't exist, we don't know.

Big- O Notation

- We now define the set

$$O(g) = \{f : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

- If $f \in O(g)$, then we say that “ f is big- O of g ” or that g *grows at least as fast as f* .
- Note that if $f \in O(g)$, then either $f \in \Theta(g)$ or $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
- Some other notation
 - $f \in \Omega(g) \Leftrightarrow g \in O(f)$.
 - $f \in o(g) \Leftrightarrow f \in O(g) \setminus \Theta(g) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.
 - $f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.

Recursion

- A recursive function is one that calls itself.
- There are two basic types of recursive functions.
 - A *linear recursion* calls itself once.
 - A *branching recursion* calls itself two or more times.
- Generally speaking, recursive algorithms should have the following two properties to be guaranteed well-defined termination.
 - They should solve an explicit **base case**.
 - Each recursive call should be made with a **smaller input size**.
- The use of recursion makes many algorithms easier to implement, but there are drawbacks.
- Recursive implementations usually use more memory and may not be quite as efficient as their nonrecursive counterparts.
- Every recursive algorithm has a nonrecursive counterpart.

Divide and Conquer

- Many common problems can be solved using a *divide-and-conquer* approach.
- This means breaking a larger problem into pieces that can be solved independently.
- The solutions to the various pieces may then have to be recombined in some way.
- Divide-and-conquer algorithms have natural implementations using branching recursions.
 - Divide: Divide the input data into smaller pieces.
 - Conquer: Call the algorithm recursively on each piece.
 - Combine: Combine the results into a solution to the original problem.

Analyzing Recursive Algorithms

- The running times of many divide and conquer algorithms can be analyzed by solving a *recurrence*.
- For an input of size n , let the running times of the divide and combine steps be $f(n)$.
- Suppose that the divide step results in a smaller pieces of size n/b (there may be some overlap).
- If $T(n)$ is the overall running time of the algorithm, then $T(n)$ must satisfy the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- The running times of other recursive algorithms also give rise to recurrences.

Analyzing Recurrences

- General methods for analyzing recurrences
 - Make a **guess** and prove that it's right (usually with induction).
 - Build a **recursion tree**.
 - Use **telescoping** (generally used for linear recursions).
 - Use the **Master Theorem** (generally used for branching recursions).
- When we analyze a recurrence, we may not get or need an exact answer.
- We may prove the running time is in $O(f)$ or $\Theta(f)$ for some simpler function f .
- When taking the ratio of two integers, it usually doesn't matter whether we round up or down.

The Master Theorem

- We can use the **Master Theorem** to analyze a divide and conquer recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

- We have to choose from one of three cases:
 1. If $f(n) \in O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
 2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \lg n)$.
 3. If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, then $T(n) \in \Theta(f(n))$.
- If f is a polynomial, then deciding which case we are in is as simple as comparing the degree of f to $\log_b a$.

Data Structures

Data Structures

- Algorithms use *data structures* to store and manipulate data during the course of execution.
- A data structure consists of a specified set of data and a set of algorithms for performing operations on the data.
- In C++, data structures have natural implementations as new *data types* (classes).
- A C++ class is composed of
 - *data members*, and
 - *member functions*.
- The *data members* are the values.
- The *member functions* are the operations to be performed on these values.
- The most important concept to remember is that of separating the *definition* (interface) from the *implementation*.

Lists

- A **list** is the most basic data structure.
- A list stores a set of elements and supports the following operations.
 - Get the number of elements in the list.
 - Get element j .
 - Set element j .
 - Add an element to the list just before element j .
 - Delete element j from the list.
- There are two basic implementations for lists
 - **arrays**
 - **linked lists**

Stacks and Queues

- A *stack* is a special kind of list in which items can only be removed in “last-in, first-out” (LIFO) order.
- A *queue* is a list in which items can only be removed in “first-in, first-out” (FIFO) order.
- The basic operations on a stack are
 - *push* a new item on the stack.
 - *pop* the most recently added item off the stack.
- The basic operations on a queue are
 - *enqueue* a new item.
 - *dequeue* the most recently added item.
- Stacks and queues can also be implemented using either arrays or linked lists.

Trees

- A *tree* is a set of items organized into a hierarchical structure.
- When organized in this way, we call the items *nodes*.
- Each node has a single designated *parent* and one or more *children*.
- There is a single designated node, called the *root*, with no parent.
- Any node with no children is called a *leaf*.
- Any node with children is called *internal*.
- A tree in which all nodes have 2 or fewer children is called a *binary tree*.
- Storing a list of items in a tree structure allows us to represent **additional relationships** among the items in the list.

Binary Tree Data Structures

- To store a tree, we need a **node** data structure supporting three basic operations.
 - **parent()**: return a pointer to the parent of a node.
 - **right()**: return a pointer to the “right” child of a node.
 - **left()**: return a pointer to the “left” child of a node.
- This allows us to **traverse** the tree and perform other operations on it.
- The **level** of a node in the tree is the number of recursive calls to **parent()** needed to reach the root.
- The **depth** of the tree is the maximum level of any of its nodes.
- A **balanced tree** is one in which all leaves are at levels k or $k - 1$, where k is the depth of the tree.

Data Structures for Storing Trees

- There are two primary data structures used for storing trees.
- **Array**
 - The root is stored in position 0 .
 - The children of the node in position i are stored in positions $2i$ and $2i + 1$.
 - This determines a unique storage location for every node in the tree and makes it easy to find a node's parent and children.
 - Using an array, the basic operations can be performed very efficiently.
 - If the tree is unbalanced or dynamic, a linked list may be better.
- **Linked List**
 - In a linked list, each item is stored along with explicit pointers to its parent and children.
 - This allows for easy addition and deletion of nodes from the tree.

Priority Queues

- A priority queue is a data structure for maintaining a list of items that have associated *priorities*.
- The usual operations are
 - **construct** a queue from a list of items.
 - **find** the item with the highest priority.
 - **insert** an item.
 - **delete** an item.
 - **change** the priority of an item.
- The most common implementation of priority queues is using a *heap*.
- A heap is a binary tree in which **the record stored at each node has a higher priority than either of its children**.

Sorting

The Sorting Problem

- The sorting problem is fundamental to the study of algorithms.
- Algorithms for sorting are used in a vast number of applications and much is known about them.
- Most often, the items to be sorted are individual *records*, usually consisting of a *key* and related *satellite data*.
- The sorting problem is defined as follows.
 - Input: A sequence of n records a_1, a_2, \dots, a_n .
 - Output: A reordering a'_1, a'_2, \dots, a'_n of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- Note that the records can be anything for which a “ \leq ” operator can be defined (usually by comparing the specified key).
- It is known that the running time of any **comparison-based** sorting algorithm is in $\Omega(n \lg n)$.

Properties of Sorting Algorithms

- In addition to running time, there are a few important properties of sorting algorithm that we may need to consider.
 - A *stable* sorting algorithm is one that leaves duplicate keys in the same relative order that they were in the original list.
 - This is an important property if you want to be able to sort on multiple keys.
 - Another important consideration is whether the algorithm sorts *in place*, i.e., does not have to allocate too much extra memory.
 - Finally, we might consider how well the algorithm performs on arrays that are already sorted, or mostly sorted.
- The sorting algorithm you choose may depend on what you expect the data to look like, e.g., is it “almost sorted.”
- The basic operations performed in sorting are **comparison** and **exchange**.
- The relative cost of these operations may also help determine the type of sort that is most appropriate.

Basic Sorting Algorithms

- Most straightforward sorting algorithms have a running time in $O(n^2)$.
- Nonadaptive algorithms perform the same sequence of steps for any input and have running times that are very consistent.
 - Selection sort
 - Bubble sort
- Adaptive algorithms can have dramatically different running times for different inputs.
 - Insertion sort: Fast for data that is “almost sorted.”
 - Quicksort: Fast for “random” data.

Commonly Used Sorting Algorithms

- **Insertion Sort**

- Very efficient for “almost sorted” data.
- Can be implemented **in place** and is **stable**.
- Slow on average.

- **Merge Sort**

- **Asymptotically optimal** and **stable**.
- Cannot be implemented **in place**.

- **Heap Sort**

- Heap sort is **asymptotically optimal** and can be implemented **in place**, but it is **unstable**.
- Heap sort is based on the construction of a **priority queue**.

- **Quicksort**

- Randomized quicksort has excellent **average case performance** ($\Theta(n \lg n)$) and can be implemented **in place**.
- However, it is **unstable** and can result in a large call stack and poor performance if not implemented carefully.

Searching

Symbol Tables and Dictionaries

- In the last few lectures, we discussed various methods for sorting a list of items by a specified key.
- We now consider further operations on such lists.
- A *symbol table* is a data structure for storing a list of items, each with a *key* and *satellite data*, supporting the following basic operations.
 - *construct* a symbol table.
 - *search* for an item (or items) having a specified key.
 - *insert* an item.
 - *remove* a specified item.
 - *count* the number of items.
 - *print* the list of items.
- Symbol tables are also called *dictionaries* because of the obvious comparison with looking up entries in a dictionary.
- Note that the keys may not have an ordering.

Additional Operations on Symbol Tables

- If the items can be ordered, e.g., by `operator<` and `operator==`, we may support the following additional operations.
 - `Sort` the items (print them in sorted order).
 - Return the `maximum` or `minimum` item.
 - `Select` the k^{th} item.
 - Return the `successor` or `predecessor` of a given item.
- We may also want to be able to `join` two symbol tables into one.
- These operations may or may not be supported in various implementations.
- The easiest implementation is using an `array`.

Binary Search Trees

- A BST can be used to implement a symbol table when the keys have an order.
- As with heaps, a binary search tree is a binary tree with additional structure.
- In a binary tree, the key value of any node is
 - greater than or equal to the key value of all nodes in its *left subtree*;
 - less than or equal to the key value of all nodes in its *right subtree*.
- With this simple structure, we can implement all functions efficiently.

Performance of BSTs

- Efficiency of the basic operations depends on the depth of the tree.
- The best case is to make the same comparisons as in binary search.
- However, this can only happen if the root of each subtree is the median element of that subtree, i.e., the tree is balanced.
- Fortunately, if keys are added at random, this should be the case “on average.”
 - Like quicksort, the average performance is very good, but worst case behavior is easy to find ([where?](#)).
 - In fact, quicksort and BSTs exhibit worst case behavior on the same inputs!
 - As with quicksort, one can show that for a random sequence of keys, the average depth of the tree is $2 \ln n \approx 1.39 \lg n$.
 - Again, the average depth is only **40% higher** than the best possible.
 - Building a binary search tree has the same running time as quicksort!

Hash Tables

- *Hash tables* are a data structure for storing a dictionary that supports only the operations
 - insert,
 - delete, and
 - search.
- Most data structures for storing dictionaries depend on using comparison and exchange to order the items.
- This limits the efficiency of certain operations (recall the lower bound on the efficiency of comparison-based sorting).
- A *hash table* is a generalization of an array that takes advantage of our ability to access an arbitrary array element in constant time.
- Using hashing, we determine where to store an item in the table (and how to find it later) without using comparison.
- This allows us to perform all the basic operations extremely efficiently.

Hash Functions

- A *hash function* is a function $h : U \rightarrow 0, \dots, M - 1$ that takes a key and converts it into an array index (called the *hash value*).
- Once we have a hash function, we can use the very efficient array-based implementation to store the table.
- A good hash function **minimizes collisions** and is **easy to compute**.
- For a “random” key, we would like the probability of each hash value to be “equally likely.”
- A simple method to hash a key x , take $x \bmod M$, where M is the size of the hash table (typically a prime number).
- This is called *modular hashing* and is a very popular form of hashing.

Resolving Collisions

- There are two primary of methods of resolving collisions.
- **Chaining**: Form a linked list of all the elements that hash to the same value.
 - Easy to implement.
 - The table never “fills up” (better for extremely dynamic tables)
 - May use more memory overall.
 - Easy to insert and delete.
- **Open Addressing**: If the hashed address is already used, use a simple rule to systematically look for an alternate.
 - Very efficient if implemented correctly.
 - When the table is nearly full, basic operations become very expensive.
 - Deleting items can be very difficult, if not impossible.
 - Once the table fills up, no more items can be added until items are deleted or the table is reallocated (expensive).

Disjoint Set Data Structure

- A disjoint set data structure is used to store a set of items that consists of a number of disjoint subsets.
- There are two basic operations we'd like to be able to perform.
 - `find(i, j)`: Are `i` and `j` in the same subset?
 - `union(i, j)`: Combine the subsets containing `i` and `j`.
- This type of data structure is sometimes called a *union-find* data structure.
- The most naive implementations of union-find are inefficient for one of the two operations.
- Making both operations efficient requires techniques that are not obvious.

Graphs

Graphs

- A *graph* is a data structure used to store connectivity relations.
- A *graph* consists of a list of items, along with a set of connections between the items.
- An undirected graph $G = (V, E)$ is composed of a set of vertices V and a set of edges $E \subseteq V \times V$ that are unordered pairs.
- A directed graph $G = (N, A)$ is composed of a set of vertices N and a set of arcs $A \subseteq V \times V$ that are ordered pairs.
- In a *weighted graph*, each edge or arc has a real number, called its *weight*, associated with it.
- Basic operations on a graph
 - Inserting an edge.
 - Deleting an edge.
 - Enumerating all edges incident to/from/on a node.
 - Searching the graph.

Graph Implementations

- To support basic graph operations, we need a method of storing the graph (an implementation.
- As with many previous data structures, there are basically two different ways to compactly represent a graph.
 - **Adjacency matrix**: An implementation based on arrays.
 - **Adjacency lists**: An implementation based on linked lists.
- We analyzed the tradeoffs between these two representations.
 - Generally, an adjacency matrix is more appropriate for dense graphs.
 - An adjacency list is more appropriate for sparse graphs.

Finding the Components of a Graph

- The most basic property of a graph we are interested in is whether two vertices are connected or not.
- This question can be answered for all vertices by finding the connected components of the graph.
- There are two basic methods for finding the components.
 - Union-find
 - * Components can be computed “on-line.”
 - * Very efficient if no other information about the graph is needed.
 - Graph search
 - * Must construct a graph representation first.
 - * Better if the graph will be queried for more information later.

Searching a Graph

- *Graph search* is a general method used to reveal various properties of a graph.
- Many algorithms are based on this general framework.
- *Graph search* consists of systematically *processing* the vertices of a graph to discover some property of the graph.
- To search a single component:
 - Choose a start vertex and add it to the list of unprocessed vertices.
 - Repeat until no vertices remain on the list.
 - * Choose a vertex v from the list of unprocessed vertices.
 - * Process v .
 - * Add all the neighbors of v to the list of unprocessed vertices.

Types of Graph Search

- Note that we have left three basic components unspecified in our description of graph search.
 - How to **determine the starting vertex** (for each component).
 - How to **process a vertex**.
 - How to **select a vertex** from the list of vertices to be processed.
- The way in which these three steps are implemented determines the overall running time of the algorithm.
- The various options result in a rich class of algorithms that can answer many interesting questions about a given graph.

Trees as Graphs

- In graph terminology, a *tree* is a connected graph with no cycles and a *forest* is a graph consisting of a collection of trees.
- Properties of trees
 - Every tree has exactly $n - 1$ edges.
 - In a tree, there is a **unique path** from any given vertex to any other vertex.
- A tree that has a specified *root vertex* is called a rooted tree.
 - In a rooted tree, there is a unique path from the root to every other vertex.
 - We can therefore uniquely define the parent of a vertex v as the vertex that immediately precedes it on the path from the root to v .
 - Hence, we are justified in thinking of trees in the way that we had previously, as a set of hierarchical relationships between the vertices.

Search Trees and Forests

- Consider searching a connected undirected graph $G = (V, E)$.
- The process of searching G can be captured by constructing a tree T called the *search tree*.
- T is constructed as the search evolves by adding an edge connecting the vertex currently being processed to any vertex not yet processed.
- This graph must be connected and acyclic, and hence is a tree.
- We can view it as a rooted tree by taking the root to be the start vertex.
- In graphs with multiple components, we can similarly obtain *search forests*.

Depth-first and Breadth-first Search

- Two common graph search algorithms choose the next node to be processed based on its depth.
- **Depth-first search (DFS)**
 - **DFS** chooses the next vertex to be processed as the vertex at maximum depth in this tree.
 - **DFS** tends to produce very deep search trees.
 - **DFS** is easy to implement, either with recursion or by using a stack.
- **Breadth-first search**
 - **BFS** chooses the next vertex to be processed as the vertex at maximum depth in this tree.
 - **BFS** results in a very shallow search tree, unlike DFS.
 - **BFS** is implemented using a queue instead of a stack to store the unprocessed nodes.

Shortest Paths

- In an unweighted graph, the path from the root node to any other vertex in the BFS search tree is a shortest path (in terms of number of edges).
- In a weighted graph, the **length of a path** is the sum of the weights of the edges encountered on the path.
- A **shortest path** between two vertices in a weighted graph is a path connecting the two vertices that is of minimum length.
- A **shortest paths tree** (SPT) is a rooted tree in which the path from the root vertex to each other vertex in the graph is a shortest such path in the original graph.
- To construct a shortest paths tree in the weighted case, we can use **Dijkstra's** Algorithm.

Algorithm Summary

- We are given a graph $G = (V, E)$ and a source node r from which we want to find shortest paths to all other nodes.
- Algorithm
 - Initialize by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes $v \in V \setminus \{r\}$.
 - Place r on the list L of unprocessed nodes.
 - While L is not empty
 - * Choose $v \in L$ such that $d(v) = \min_{u \in L} d(u)$.
 - * For each neighbor x of v , set $d(x) = \min\{d(x), d(v) + w_{\{v,x\}}\}$.
- When the algorithm is completed, we will have $d(v) = \delta(v)$ for all $v \in V$ and the search tree will be a shortest paths tree.
- This algorithm can be implemented using a priority queue to store the list of unprocessed nodes and has a running time of $O(m \lg n)$ if the graph is connected.

Spanning Trees

- Given a connected undirected graph $G = (V, E)$, a *spanning tree* T of G is a subgraph that is a tree and whose vertex set is all of V .
- Every *minimal* connected subgraph is a spanning tree (and vice versa).
- In other words, a subgraph is a spanning tree if and only if it is connected and removing any edge will disconnect it.
- If $T \subseteq E$ is a spanning tree of G , the *weight* of T is

$$\sum_{e \in T} w_e$$

- The *minimum weight spanning tree* (MST) problem is that of finding, among all spanning trees of G , one that has minimum weight.

Prim's Algorithm for Finding an MST

- We are given a connected undirected weighted graph $G = (V, E)$ and we want to find an MST of G .
- Prim's Algorithm
 - Arbitrarily choose a source node r .
 - Initialize by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes $v \in V \setminus \{r\}$.
 - Place r on the list L of unprocessed nodes.
 - While L is not empty
 - * Choose $v \in L$ such that $d(v) = \min_{u \in L} d(u)$.
 - * For each neighbor x of v , set $d(x) = \min\{d(x), w_{\{v,x\}}\}$.

Another Approach

- Another class of algorithms, called *greedy algorithms* takes the approach of building an MST edge by edge.
- Let's assume that we have a set of edges T that satisfies the property that T can be extended to an MST.
- Question: What edges can we add to T to maintain the property?
- If we add an edge that is a minimum edge crossing some cut S , this property is maintained.
- Rationale: In any connected graph, there must be an edge crossing each cut in the graph.
- We will call such edge a *safe edge* if it also doesn't create a cycle when added to T .
- How do we find such an edge?
 - **Prim's Algorithm** simply considers the cut S consisting of nodes that have already been processed.
 - At each step, we add the minimum edge crossing that cut.
 - There are other possibilities, however.

Kruskal's Algorithm

- **Kruskal's Algorithm** takes a more global view.
- At each step, we consider *all edges* that do not form a cycle when added to the current set T .
- The minimum such edge is guaranteed to be **safe** (why?).
- As edges are added, we will keep track of the current set of components using a **union-find** data structure.
- At each step, we'll add the cheapest edge to T that doesn't connect two nodes currently in the same component.
- Implementing **Kruskal's Algorithm**
 - Before beginning, sort the edges by weight and set $T = \emptyset$.
 - While there are unexamined edges on the list
 - * Call the **find()** operation on the endpoints of each edge until an edge e is found for whose endpoints are in different components.
 - * After adding e to T , call the **union()** operation to combine the components containing its endpoints into a single component.

Data Structures for Digraphs

- Data structures for digraphs are similar to those for undirected graphs.
- As before, there are two basic choices
 - Adjacency matrix
 - Adjacency lists
- These are implemented in similar fashion, except that
 - In the case of an adjacency matrix, the matrix is no longer **symmetric**.
 - In the case of an adjacency list, each arc appears only on the adjacency list of its **tail vertex**.

Graph Search for Digraphs

- Graph search doesn't have the same interpretation in the directed case because we cannot use it directly to find the (strongly) connected components.
- Graph search from vertex r :
 - Add r to the list of unprocessed vertices.
 - Repeat until no vertices remain on the list.
 - * Choose a vertex v from the list of unprocessed vertices.
 - * Process v .
 - * For each arch incident from v , add its head to the list of unprocessed vertices.
- As before, this graph search process results in construction of a **directed search tree** in which there is a path from r to each other vertex.
- Such a directed tree is said to be **directed away from r** .
- This graph search algorithm can easily be adapted to find the shortest directed paths from r to each other vertex.
- It can also be used to find a minimum spanning tree directed away from r .

Directed Acyclic Graphs

- Often, directed graphs are used to represent precedence relations.
- Such *precedence graphs* should be both *directed* and *acyclic*.
- Given a directed acyclic graph (DAG), one thing we would like to be able to do is determine an ordering of vertices that obeys the precedence relations.
- This is called a *topological sort*.
- Given a DAG, DFS can be used to perform a topological sort.
 - Each vertex is added to the front of a linked list after all of its neighbors have been processed.
 - The resulting linked list is a topological sort.

Numerical Analysis

Floating-point Numbers

- There are basically three schemes by which real numbers can be approximated in a computer.
 - As a rational number.
 - Using fixed point representation.
 - Using floating point representation.
- In modern computers, real numbers are approximated using a *floating point* representation.
- The floating-point numbers F are a subset of the real numbers.
- A floating point number consists of three parts:
 - The sign
 - The exponent
 - The mantissa

Floating-point Arithmetic

- Arithmetic with floating point numbers is different than regular arithmetic.
- This is because after each operation, the answer must be rounded off.
- The biggest problems occur when numbers on different scales appear in the same calculation.
- Example
 - Assume 10 digit precision
 - $(10^{-10} + 1) - 1 = 0$
 - $10^{-10} + (1 - 1) = 10^{-10}$
- The example show that floating point operations do not have the same properties as familiar arithmetic operations.

Vectors and Matrices

- Vectors and matrices are constructs that arise naturally in many systems engineering applications.
- Operating on vectors and matrices requires numerical algorithms.
- An $m \times n$ *matrix* is an array of mn real numbers:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- A is said to have n *columns* and m *rows*.
- An n -*dimensional column vector* is a matrix with one column.
- An n -*dimensional row vector* is a matrix with one row .
- By default, a *vector* is considered a column vector.

Storing Vectors and Matrices

- The *density* of a matrix is the percentage of entries that are nonzero.
- *Dense* vectors can simply be stored in an array.
- *Dense* matrices can be stored in a 2-dimensional array.
- Matrices that arise in practice, however, are typically *sparse*.
- Sparse matrices can be stored using a strategy similar adjacency lists using three vectors:
 - The first vector stores the locations of the nonzero entries in each column.
 - The second vector stores the actual values corresponding to each one of those locations.
 - The third vector stores the location in the first two matrices of the entries corresponding to each column.

Multiplying Matrices

- A straightforward method for multiplying 2×2 matrices can be generalized to yield a simple recursive algorithm.
- Divide each matrix into $\frac{n}{2} \times \frac{n}{2}$ submatrices.

$$\begin{bmatrix} R & S \\ T & U \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

- To calculate the answer, we have:

$$R = AE + BG$$

$$S = AF + BH$$

$$T = CE + DG$$

$$U = CF + DH$$

The LUP Decomposition

- An **LUP** decomposition consists of three $n \times n$ matrices L , U , and P such that

$$PA = LU$$

where

- L is upper triangular.
 - U is lower triangular with 1's on the diagonal.
 - P is a permutation matrix.
- Once we have an LUP decomposition, we can use it to solve the system $Ax = b$.
 - First solve the system $Ly = Pb$ (forward substitution).
 - Then solve the system $Ux = y$ (backward substitution).
 - Once the factorization is known, this takes only $O(n^2)$ steps, as opposed to $O(n^3)$ for Gaussian elimination.

Finding the LU Decomposition (cont.)

- The following equation shows how to find an LU factorization recursively and “in place.”

$$A = \begin{bmatrix} 1 & 0 \\ v/a_{11} & I \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{bmatrix} \quad (1)$$

$$= \begin{bmatrix} 1 & 0 \\ v/a_{11} & I \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & L'U' \end{bmatrix} \quad (2)$$

$$= \begin{bmatrix} 1 & 0 \\ v/a_{11} & L' \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & U' \end{bmatrix} \quad (3)$$

- The element a_{11} is called the *pivot element*.
- Note that the above decomposition method fails whenever the pivot element is zero.
- In this case, we can permute the rows of A to obtain a new pivot element (this is where the matrix P comes from).
- The total running time is $O(n^3)$, the same as Gaussian elimination.

Cryptography

- Cryptography is the study of methods for sending messages in an encoded form that can (hopefully) only be interpreted by the intended recipient.
- The original message is said to be in *plaintext* and the encoded message is said to be in *ciphertext*.
- Let \mathcal{P} be the set of all plaintext messages and \mathcal{C} be the set of all encrypted messages.
- A *cryptosystem* is a one-to-one mapping $f : \mathcal{P} \rightarrow \mathcal{C}$, whose inverse maps \mathcal{C} back to \mathcal{P} .
- Most cryptosystems work by dividing the original message into *message units*, which are then individually enciphered.
- A message unit is typically defined to be a block of k letters for some positive k .
- For ease of defining the transformation, we can convert each message unit to a unique integer by interpreting it as a k -digit number base N .

Public Key Encryption

- For a cryptosystem to be useful, it has to be possible to easily encode and decode.
- Until about 25 years ago, all known cryptosystems had the property that if you knew the encoding key, you could easily derive the decoding key.
- This creates problems when trying to send an encrypted message to someone without prior arrangement.
- *Public key encryption* is an attempt to overcome this shortcoming.
- Public key systems are based on the concept of a *trapdoor function*.
- A *trapdoor function* is one which is easy to compute but “difficult” to invert without additional information.
- Using a trapdoor function to do the encoding makes it difficult to discover the decoding key from the encoding key.
- This allows the establishment of secure communications between parties without prior arrangement.
- It also provides an easy method for digitally signing electronic transmissions.

RSA Public Key Encryption Algorithm

- The RSA algorithm is used almost universally to encrypt data on the Internet.
- If you have ever visited a secure site on the Internet, you have used RSA encryption.
- Procedure for creating public and private keys.
 - Randomly choose two large prime numbers p and q such that $p \neq q$.
 - Compute $n = pq$.
 - Select an odd integer e that is relatively prime to $\phi(n) = (p - 1)(q - 1) = n + 1 - p - q$.
 - Compute d as the multiplicative inverse of e modulo $\phi(n)$.
 - The pair (e, n) is the **public key**.
 - The pair (d, n) is the **private key**.
- The encoding function is $f_E(P) = P^e \pmod n$.
- The decoding function is $f_D(C) = C^d \pmod n$.

Implementing RSA Encryption

- The RSA encryption algorithm is easy to state, but implementing it efficiently can be challenging.
- Security of the method is based on the difficulty of factoring large integers.
- To generate the public key, one must choose two large prime numbers.
- This is usually accomplished by generating random integers until two are found that can be proven prime.
- A typical method for proving a number prime involves modular exponentiation, which can be implemented efficiently.
- Once the public key is determined, the private key can be calculated using the extended version of *Euclid's Algorithm*.
- One of the biggest challenges is that the numbers involved in implementing this algorithm are HUGE.
- New data types must be implemented to store these numbers.
- Ultimately, this can all be implemented efficiently.